Next: Structure and Local Search

Next: Structure and Local Search


Efficient Solution of CSPs
Next: Structure and Local Search


Efficient Solution of CSPs
Local Search
Reminder: CSPs

Reminder: CSPs


CSPs:
- Variables
- Domains
- Constraints
- Implicit (provide code to compute)
- Explicit (provide a list of the legal tuples)
- Unary / Binary / N-ary
Reminder: CSPs


CSPs:
- Variables
- Domains
- Constraints
- Implicit (provide code to compute)
- Explicit (provide a list of the legal tuples)
- Unary / Binary / N-ary

Goals:
- Here: find any solution
- Also: find all, find best, etc.
Structure

Problem Structure


Extreme case: independent subproblems

Example: Tasmania and mainland do not interact

Independent subproblems are identifiable as connected components of constraint graph

Suppose a graph of $n$ variables can be broken into subproblems of only $c$ variables:

Worst-case solution cost is $O\left(\frac{n}{c}\right)^{2^c}$, linear in $n$

E.g., $n = 80$, $d = 2$, $c = 20$

$2^{80} = 4$ billion years at 10 million nodes/sec

$\left(\frac{4}{2}\right)^{2^{20}} = 0.4$ seconds at 10 million nodes/sec
Extreme case: independent subproblems
- Example: Tasmania and mainland do not interact
Extreme case: independent subproblems

- Example: Tasmania and mainland do not interact
Extreme case: independent subproblems

- Example: Tasmania and mainland do not interact

Independent subproblems are identifiable as connected components of constraint graph
Problem Structure


Extreme case: independent subproblems
* Example: Tasmania and mainland do not interact

Independent subproblems are identifiable as connected components of constraint graph

Suppose a graph of \( n \) variables can be broken into subproblems of only \( c \) variables:
* Worst-case solution cost is \( O((n/c)(d^c)) \), linear in \( n \)
Extreme case: independent subproblems

- Example: Tasmania and mainland do not interact

Independent subproblems are identifiable as connected components of constraint graph

Suppose a graph of $n$ variables can be broken into subproblems of only $c$ variables:

- Worst-case solution cost is $O((n/c)(d^c))$, linear in $n$
- E.g., $n = 80$, $d = 2$, $c = 20$
Problem Structure


Extreme case: independent subproblems
- Example: Tasmania and mainland do not interact

Independent subproblems are identifiable as connected components of constraint graph

Suppose a graph of $n$ variables can be broken into subproblems of only $c$ variables:
- Worst-case solution cost is $O((n/c)(d^c))$, linear in $n$
  - E.g., $n = 80$, $d = 2$, $c = 20$
  - $280 = 4$ billion years at 10 million nodes/sec
Extreme case: independent subproblems
- Example: Tasmania and mainland do not interact

Independent subproblems are identifiable as connected components of constraint graph

Suppose a graph of $n$ variables can be broken into subproblems of only $c$ variables:
- Worst-case solution cost is $O((n/c)(d^c))$, linear in $n$
- E.g., $n = 80$, $d = 2$, $c = 20$
- $280 = 4$ billion years at 10 million nodes/sec
- $(4)(2^{20}) = 0.4$ seconds at 10 million nodes/sec
Tree-Structured CSPs

Theorem: if the constraint graph has no loops, the CSP can be solved in $O(d^2)$ time. Compare to general CSPs, where worst-case time is $O(d^n)$. This property also applies to probabilistic reasoning (later): an example of the relation between syntactic restrictions and the complexity of reasoning.
Tree-structured CSPs

Theorem: if the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time.

Compare to general CSPs, where worst-case time is $O(d^n)$. 
Theorem: if the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time. Compare to general CSPs, where worst-case time is $O(d^n)$. This property also applies to probabilistic reasoning (later): an example of the relation between syntactic restrictions and the complexity of reasoning.
Tree-Structured CSPs


Algorithm for tree-structured CSPs:

- **Order:** Choose root variable and order variables so that parent precedes children.
- **Remove backward:** For $i = n$ down to 2, apply RemoveInconsistent(Parent($X_i$), $X_i$).
- **Assign forward:** For $i = 1$ up to $n$, assign $X_i$ consistently with Parent($X_i$).

**Runtime:** $O(nd^2)$ (why?)
Tree-Structured CSPs


Algorithm for tree-structured CSPs:
Algorithm for tree-structured CSPs:

- **Order:**

```latex
\text{Choose root variable and order variables so that parent precedes children}
```

```latex
\text{Remove backward: For } i = n \rightarrow 2, \text{ apply } \text{RemoveInconsistent(Parent(X_i)}, X_i)\text{)}
```

```latex
\text{Assign forward: For } i = 1 \rightarrow n, \text{ assign } X_i \text{ consistently with Parent(X_i)}
```

**Runtime:** $O(n^2)$ (why?)
Tree-Structured CSPs


Algorithm for tree-structured CSPs:
- Order: Choose root variable
Algorithm for tree-structured CSPs:
- Order: Choose root variable
  and order variables so that parent precedes children
Tree-Structured CSPs


Algorithm for tree-structured CSPs:
- **Order:** Choose root variable
  and order variables so that parent precedes children
- **Remove backward:**

![Tree-Structured CSP Diagram](http://bit.ly/3GEMok7)
Algorithm for tree-structured CSPs:
- **Order**: Choose root variable and order variables so that parent precedes children
- **Remove backward**
Algorithm for tree-structured CSPs:
- Order: Choose root variable and order variables so that parent precedes children
- Remove backward:
  \[ \text{For } i = n : 2, \text{ apply RemoveInconsistent(Parent}(X_i),X_i) \]
Algorithm for tree-structured CSPs:

- Order: Choose root variable and order variables so that parent precedes children
- Remove backward: For $i = n : 2$, apply RemoveInconsistent(Parent($X_i$), $X_i$)
- Assign forward:
Tree-Structured CSPs

Algorithm for tree-structured CSPs:

- Order: Choose root variable and order variables so that parent precedes children

- Remove backward:
  For $i = n : 2$, apply RemoveInconsistent(Parent($X_i$), $X_i$)

- Assign forward:
  For $i = 1 : n$, assign $X_i$ consistently with Parent($X_i$)
Algorithm for tree-structured CSPs:
- Order: Choose root variable and order variables so that parent precedes children
- Remove backward: For $i = n : 2$, apply RemoveInconsistent(Parent($X_i$), $X_i$)
- Assign forward: For $i = 1 : n$, assign $X_i$ consistently with Parent($X_i$)

Runtime: $O(nd^2)$ (why?)
Tree-Structured CSPs

Claim 1: After backward pass, all root-to-leaf arcs are consistent
Proof: Each \( X \leftarrow Y \) was made consistent at one point and Y's domain could not have been reduced thereafter (because Y's children were processed before Y).

Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
Proof: Induction on position

Why doesn't this algorithm work with cycles in the constraint graph?
Note: we'll see this basic idea again with Bayes' nets
Claim 1: After backward pass, all root-to-leaf arcs are consistent

Proof: Each $X \leftarrow Y$ was made consistent at one point and $Y$'s domain could not have been reduced thereafter (because $Y$'s children were processed before $Y$)

Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack

Proof: Induction on position

Why doesn't this algorithm work with cycles in the constraint graph?

Note: we'll see this basic idea again with Bayes' nets
Claim 1: After backward pass, all root-to-leaf arcs are consistent

Why doesn't this algorithm work with cycles in the constraint graph?

Note: we'll see this basic idea again with Bayes' nets
Claim 1: After backward pass, all root-to-leaf arcs are consistent

Proof: Each $X \leftarrow Y$ was made consistent at one point and Y’s domain could not have been reduced thereafter (because Y’s children were processed before Y)

Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack

Proof: Induction on position

Why doesn’t this algorithm work with cycles in the constraint graph?

Note: we’ll see this basic idea again with Bayes’ nets
Tree-Structured CSPs

Claim 1: After backward pass, all root-to-leaf arcs are consistent

Proof: Each \( X \leftarrow Y \) was made consistent at one point and Y’s domain could not have been reduced thereafter (because Y’s children were processed before Y)

Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack
Claim 1: After backward pass, all root-to-leaf arcs are consistent

Proof: Each $X \leftarrow Y$ was made consistent at one point and Y’s domain could not have been reduced thereafter (because Y’s children were processed before Y)

Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack

Proof: Induction on position
Tree-Structured CSPs

Claim 1: After backward pass, all root-to-leaf arcs are consistent

Proof: Each $X \leftarrow Y$ was made consistent at one point and Y’s domain could not have been reduced thereafter (because Y’s children were processed before Y)

Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack

Proof: Induction on position

Why doesn’t this algorithm work with cycles in the constraint graph?
Claim 1: After backward pass, all root-to-leaf arcs are consistent

Proof: Each $X \leftarrow Y$ was made consistent at one point and Y’s domain could not have been reduced thereafter (because Y’s children were processed before Y)

Claim 2: If root-to-leaf arcs are consistent, forward assignment will not backtrack

Proof: Induction on position

Why doesn’t this algorithm work with cycles in the constraint graph?

Note: we’ll see this basic idea again with Bayes’ nets
Improving Structure
Nearly Tree-Structured CSPs

Conditioning: instantiate a variable, prune its neighbors' domains
Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
Cutset size $c$ gives runtime $O\left((d^c(n-c))d^2\right)$, very fast for small $c$
Nearly Tree-Structured CSPs

Conditioning: instantiate a variable, prune its neighbors’ domains
Nearly Tree-Structured CSPs

Conditioning: instantiate a variable, prune its neighbors’ domains
Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree
Nearly Tree-Structured CSPs

Conditioning: instantiate a variable, prune its neighbors’ domains

Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

Cutset size $c$ gives runtime $O((d^c)(n - c)d^2)$, very fast for small $c$
Cutset Conditioning

1. Choose a cutset.
2. Instantiate the cutset (all possible ways).
3. Compute residual CSP for each assignment.
4. Solve the residual CSPs (tree structured).
Cutset Conditioning

Choose a cutset.
Cutset Conditioning

Choose a cutset.

Instantiate the cutset (all possible ways).
Cutset Conditioning

Choose a cutset.

Instantiate the cutset (all possible ways).

Compute residual CSP for each assignment.
Cutset Conditioning

Choose a cutset.

Instantiate the cutset (all possible ways).

Compute residual CSP for each assignment.

Solve the residual CSPs (tree structured).
Cutset Quiz

Find the smallest cutset for the graph below.
Find the smallest cutset for the graph below.
Find the smallest cutset for the graph below.
Tree Decomposition*

WA - NT - Q
SA - NSW - V
T

Agree: $(M_1, M_2) \in \{(WA = r, SA = g, NT = b), (SA = g, NT = b, Q = r)\}, \ldots$
Tree Decomposition*

Idea: create a tree-structured graph of mega-variables
Tree Decomposition*

Idea: create a tree-structured graph of mega-variables

Each mega-variable encodes part of the original CSP
Tree Decomposition*

Idea: create a tree-structured graph of mega-variables

Each mega-variable encodes part of the original CSP

Subproblems overlap to ensure consistent solutions
Tree Decomposition*

Idea: create a tree-structured graph of mega-variables

Each mega-variable encodes part of the original CSP

Subproblems overlap to ensure consistent solutions
Tree Decomposition*

Idea: create a tree-structured graph of mega-variables

Each mega-variable encodes part of the original CSP

Subproblems overlap to ensure consistent solutions
Tree Decomposition

Idea: create a tree-structured graph of mega-variables

Each mega-variable encodes part of the original CSP

Subproblems overlap to ensure consistent solutions
Tree Decomposition*

Idea: create a tree-structured graph of mega-variables

Each mega-variable encodes part of the original CSP

Subproblems overlap to ensure consistent solutions
Tree Decomposition*

Idea: create a tree-structured graph of mega-variables

Each mega-variable encodes part of the original CSP

Subproblems overlap to ensure consistent solutions

\[
\{(WA = r, NT = g, SA = b), \ldots\}
\]
Tree Decomposition*

Idea: create a tree-structured graph of mega-variables

Each mega-variable encodes part of the original CSP

Subproblems overlap to ensure consistent solutions

\{(WA = r, NT = g, SA = b), \ldots\}

\{(NT = r, SA = g, Q = b), \ldots\}
Tree Decomposition*

Idea: create a tree-structured graph of mega-variables

Each mega-variable encodes part of the original CSP

Subproblems overlap to ensure consistent solutions

\[
\{(WA = r, NT = g, SA = b), \ldots\} \quad \{(NT = r, SA = g, Q = b), \ldots\} \quad \{(Q = r, NSW = g, V = b), \ldots\}
\]
Tree Decomposition*

Idea: create a tree-structured graph of mega-variables
Each mega-variable encodes part of the original CSP
Subproblems overlap to ensure consistent solutions

Agree:

\{
(WA = r, NT = g, SA = b), \ldots
\}

\{
(NT = r, SA = g, Q = b), \ldots
\}
Tree Decomposition*

Idea: create a tree-structured graph of mega-variables

Each mega-variable encodes part of the original CSP

Subproblems overlap to ensure consistent solutions

Agree:

\[(M_1, M_2) \in \{((WA = r, SA = g, NT = b), (SA = g, NT = b, Q = r)), \ldots\}\]
Iterative Improvement
Iterative Algorithms for CSPs

Local search methods typically work with “complete” states, i.e., all variables assigned.

To apply to CSPs:
- Take an assignment with unsatisfied constraints
- Operators reassign variable values

No fringe! Live on the edge.

Algorithm: While not solved,
- Variable selection: randomly select any conflicted variable
- Value selection: min-conflicts heuristic: Choose a value that violates the fewest constraints
- I.e., hill climb with $h(n) =$ total number of violated constraints
Local search methods typically work with “complete” states, i.e., all variables assigned
Iterative Algorithms for CSPs

Local search methods typically work with “complete” states, i.e., all variables assigned

To apply to CSPs:
• Take an assignment with unsatisfied constraints
Iterative Algorithms for CSPs

Local search methods typically work with “complete” states, i.e., all variables assigned.

To apply to CSPs:
- Take an assignment with unsatisfied constraints
- Operators reassign variable values

Algorithm: While not solved,
- Variable selection: randomly select any conflicted variable
- Value selection: min-conflicts heuristic: choose a value that violates the fewest constraints
  - i.e., hill climb with $h(n) = \text{total number of violated constraints}$
Iterative Algorithms for CSPs

Local search methods typically work with “complete” states, i.e., all variables assigned

To apply to CSPs:

- Take an assignment with unsatisfied constraints
- Operators reassign variable values

\[ \neq \]
Iterative Algorithms for CSPs

Local search methods typically work with “complete” states, i.e., all variables assigned

To apply to CSPs:

- Take an assignment with unsatisfied constraints
- Operators reassign variable values

\[ \neq \]

Algorithm: While not solved,
- Variable selection: randomly select any conflicted variable
- Value selection: min-conflicts heuristic:
  - Choose a value that violates the fewest constraints
  - I.e., hill climb with \( h(n) = \text{total number of violated constraints} \)
Local search methods typically work with “complete” states, i.e., all variables assigned

To apply to CSPs:
- Take an assignment with unsatisfied constraints
- Operators reassign variable values

- No fringe! Live on the edge.
Iterative Algorithms for CSPs

Local search methods typically work with “complete” states, i.e., all variables assigned

To apply to CSPs:
- Take an assignment with unsatisfied constraints
- Operators reassign variable values

- No fringe! Live on the edge.
Iterative Algorithms for CSPs

Local search methods typically work with “complete” states, i.e., all variables assigned.

To apply to CSPs:
- Take an assignment with unsatisfied constraints
- Operators reassign variable values

No fringe! Live on the edge.

Algorithm: While not solved,
Iterative Algorithms for CSPs

Local search methods typically work with “complete” states, i.e., all variables assigned.

To apply to CSPs:
- Take an assignment with unsatisfied constraints
- Operators reassign variable values
  - No fringe! Live on the edge.

Algorithm: While not solved,
- Variable selection: randomly select any conflicted variable

\[ \neq \]

\[ \neq \]
Local search methods typically work with “complete” states, i.e., all variables assigned

To apply to CSPs:
- Take an assignment with unsatisfied constraints
- Operators reassign variable values

No fringe! Live on the edge.

Algorithm: While not solved,
- Variable selection: randomly select any conflicted variable
- Value selection: min-conflicts heuristic:
Iterative Algorithms for CSPs

Local search methods typically work with “complete” states, i.e., all variables assigned.

To apply to CSPs:
- Take an assignment with unsatisfied constraints
- Operators reassign variable values

- No fringe! Live on the edge.

Algorithm: While not solved,
- Variable selection: randomly select any conflicted variable
- Value selection: min-conflicts heuristic:
  - Choose a value that violates the fewest constraints
Iterative Algorithms for CSPs

Local search methods typically work with “complete” states, i.e., all variables assigned

To apply to CSPs:
- Take an assignment with unsatisfied constraints
- Operators reassign variable values

• No fringe! Live on the edge.

Algorithm: While not solved,
- Variable selection: randomly select any conflicted variable
- Value selection: min-conflicts heuristic:
  - Choose a value that violates the fewest constraints
  - I.e., hill climb with $h(n) = \text{total number of violated constraints}$
Iterative Algorithms for CSPs

Local search methods typically work with “complete” states, i.e., all variables assigned

To apply to CSPs:
- Take an assignment with unsatisfied constraints
- Operators reassign variable values

- No fringe! Live on the edge.

Algorithm: While not solved,
- Variable selection: randomly select any conflicted variable
- Value selection: min-conflicts heuristic:
  - Choose a value that violates the fewest constraints
  - I.e., hill climb with \( h(n) = \) total number of violated constraints
Example: 4-Queens

States: 4 queens in 4 columns (4^4 = 256 states)

Operators: move queen in column

Goal test: no attacks

Evaluation: c(n) = number of attacks

Demo: n-queens – iterative improvement (L5D1) Demo: coloring – iterative improvement
Example: 4-Queens

States: 4 queens in 4 columns (4^4 = 256 states)
Example: 4-Queens

States: 4 queens in 4 columns \((4^4 = 256 \text{ states})\)

Operators: move queen in column

States: 4 queens in 4 columns \((4^4 = 256 \text{ states})\)

Operators: move queen in column
Example: 4-Queens

States: 4 queens in 4 columns (4^4 = 256 states)
Operators: move queen in column
Goal test: no attacks
Example: 4-Queens

States: 4 queens in 4 columns ($4^4 = 256$ states)
Operators: move queen in column
Goal test: no attacks
Evaluation: $c(n) =$ number of attacks
Example: 4-Queens

States: 4 queens in 4 columns \((4^4 = 256\) states)  
Operators: move queen in column  
Goal test: no attacks  
Evaluation: \(c(n) = \) number of attacks

Demo: n-queens – iterative improvement (L5D1) Demo: coloring – iterative improvement
Video of Demo Iterative Improvement – n Queens
Video of Demo Iterative Improvement – Coloring
Performance of Min-Conflicts

Given random initial state, can solve n-queens in almost linear time for arbitrary n with high probability (e.g., n = 10,000,000)!

The same appears to be true for any randomly-generated CSP except in a narrow range of the ratio
Performance of Min-Conflicts

Given random initial state, can solve n-queens in almost linear time for arbitrary n with high probability (e.g., n = 10,000,000)!
Performance of Min-Conflicts

Given random initial state, can solve $n$-queens in almost linear time for arbitrary $n$ with high probability (e.g., $n = 10,000,000$)!

The same appears to be true for any randomly-generated CSP except in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$
Summary: CSPs

CSPs are a special kind of search problem:
- States are partial assignments
- Goal test defined by constraints

Basic solution: backtracking search

Speed-ups:
- Ordering
- Filtering
- Structure

Iterative min-conflicts is often effective in practice
Summary: CSPs

CSPs are a special kind of search problem:
- States are partial assignments
CSPs are a special kind of search problem:
- States are partial assignments
- Goal test defined by constraints
CSPs are a special kind of search problem:
- States are partial assignments
- Goal test defined by constraints
Summary: CSPs

CSPs are a special kind of search problem:
- States are partial assignments
- Goal test defined by constraints

Basic solution: backtracking search
CSPs are a special kind of search problem:
- States are partial assignments
- Goal test defined by constraints

Basic solution: backtracking search

Speed-ups:
- Ordering
Summary: CSPs

CSPs are a special kind of search problem:
- States are partial assignments
- Goal test defined by constraints

Basic solution: backtracking search

Speed-ups:
- Ordering
- Filtering
CSPs are a special kind of search problem:
- States are partial assignments
- Goal test defined by constraints

Basic solution: backtracking search

Speed-ups:
- Ordering
- Filtering
- Structure
CSPs are a special kind of search problem:
- States are partial assignments
- Goal test defined by constraints

Basic solution: backtracking search

Speed-ups:
- Ordering
- Filtering
- Structure

Iterative min-conflicts is often effective in practice
Summary: CSPs

CSPs are a special kind of search problem:
- States are partial assignments
- Goal test defined by constraints

Basic solution: backtracking search

Speed-ups:
- Ordering
- Filtering
- Structure

Iterative min-conflicts is often effective in practice
Local Search
Local Search

Tree search keeps unexplored alternatives on the fringe (ensures completeness)

Local search: improve a single option until you can't make it better (no fringe!)

New successor function: local changes.

Generally much faster and more memory efficient (but incomplete and suboptimal)
Local Search

Tree search keeps unexplored alternatives on the fringe (ensures completeness)
Local Search

Tree search keeps unexplored alternatives on the fringe (ensures completeness)

Local search: improve a single option until you can’t make it better (no fringe!)
Local Search

Tree search keeps unexplored alternatives on the fringe (ensures completeness)

Local search: improve a single option until you can’t make it better (no fringe!)
Local Search

Tree search keeps unexplored alternatives on the fringe (ensures completeness)

Local search: improve a single option until you can’t make it better (no fringe!)

New successor function: local changes.
Local Search

Tree search keeps unexplored alternatives on the fringe (ensures completeness)

Local search: improve a single option until you can’t make it better (no fringe!)

New successor function: local changes.

Generally much faster and more memory efficient (but incomplete and suboptimal)
Hill Climbing

Simple, general idea:
- Start wherever
- Repeat: move to the best neighboring state
- If no neighbors better than current, quit

What's bad about this approach?
- Complete?
- Optimal?

What's good about it?
Hill Climbing

Simple, general idea:
- Start wherever
Hill Climbing

Simple, general idea:
- Start wherever
- Repeat: move to the best neighboring state
Hill Climbing

Simple, general idea:
- Start wherever
- Repeat: move to the best neighboring state
- If no neighbors better than current, quit
Hill Climbing

Simple, general idea:
- Start wherever
- Repeat: move to the best neighboring state
- If no neighbors better than current, quit
Hill Climbing

Simple, general idea:
- Start wherever
- Repeat: move to the best neighboring state
- If no neighbors better than current, quit

What’s bad about this approach?
- Complete?
Hill Climbing

Simple, general idea:
- Start wherever
- Repeat: move to the best neighboring state
- If no neighbors better than current, quit

What’s bad about this approach?
- Complete?
- Optimal?
Hill Climbing

Simple, general idea:
- Start wherever
- Repeat: move to the best neighboring state
- If no neighbors better than current, quit

What’s bad about this approach?
- Complete?
- Optimal?
Hill Climbing

Simple, general idea:
- Start wherever
- Repeat: move to the best neighboring state
- If no neighbors better than current, quit

What’s bad about this approach?
- Complete?
- Optimal?

What’s good about it?
Hill Climbing Diagram

- Objective function
- Global maximum
- Shoulder
- Local maximum
- "Flat" local maximum
- Current state
- State space
Hill Climbing Quiz

Starting from X, where do you end up?

Starting from Y, where do you end up?

Starting from Z, where do you end up?
Hill Climbing Quiz

Starting from X, where do you end up?
Hill Climbing Quiz

Starting from X, where do you end up?
Starting from Y, where do you end up?
Hill Climbing Quiz

Starting from X, where do you end up?
Starting from Y, where do you end up?
Starting from Z, where do you end up?
Simulated Annealing

**function** SIMULATED-ANNEALING(*problem*, *schedule*) **returns** a solution state

**inputs:** *problem*, a problem

*schedule*, a mapping from time to “temperature”

**local variables:** *current*, a node

*next*, a node

*T*, a “temperature” controlling prob. of downward steps

\[
\text{current} \leftarrow \text{MAKE-NODE} (\text{INITIAL-STATE}[*problem*])
\]

**for** \( t \leftarrow 1 \) **to** \( \infty \) **do**

\[
T \leftarrow \text{schedule}[t]
\]

**if** \( T = 0 \) **then return** *current*

\[
\text{next} \leftarrow \text{a randomly selected successor of } \text{current}
\]

\[
\Delta E \leftarrow \text{VALUE}[\text{next}] - \text{VALUE}[\text{current}]
\]

**if** \( \Delta E > 0 \) **then** *current* \( \leftarrow \) *next*

**else** *current* \( \leftarrow \) *next* only with probability \( e^{\Delta E / T} \)
Simulated Annealing

function SIMULATED-ANNEALING(problem, schedule) returns a solution state
inputs: problem, a problem
        schedule, a mapping from time to “temperature”
local variables: current, a node
                next, a node
                T, a “temperature” controlling prob. of downward steps

current ← MAKE-NODE(INITIAL-STATE[problem])
for t ← 1 to ∞ do
    T ← schedule[t]
    if T = 0 then return current
    next ← a randomly selected successor of current
    ΔE ← VALUE[next] − VALUE[current]
    if ΔE > 0 then current ← next
    else current ← next only with probability $e^{ΔE/T}$

Idea: Escape local maxima by allowing downhill moves
- But make them rarer as time goes on
Simulated Annealing

\[ p(x) \propto e^{\frac{E(x)}{kT}} \]
Simulated Annealing

Theoretical guarantee:
- Stationary distribution: \( p(x) \propto e^{E(x)/kT} \)
Simulated Annealing

Theoretical guarantee:
- Stationary distribution: $p(x) \propto e^{E(x)/kT}$
- If $T$ decreased slowly enough,
Simulated Annealing

Theoretical guarantee:
- Stationary distribution: \( p(x) \propto e^{E(x)/kT} \)
- If T decreased slowly enough, will converge to optimal state!
Simulated Annealing

Theoretical guarantee:
- Stationary distribution: $p(x) \propto e^{E(x)/kT}$
- If $T$ decreased slowly enough, will converge to optimal state!
Simulated Annealing

Theoretical guarantee:

- Stationary distribution: \( p(x) \propto e^{E(x)/kT} \)
- If \( T \) decreased slowly enough, will converge to optimal state!

Is this an interesting guarantee?
Simulated Annealing

\[ p(x) \propto e^{\frac{E(x)}{kT}} \]

Theoretical guarantee:
- Stationary distribution: \( p(x) \propto e^{E(x)/kT} \)
- If \( T \) decreased slowly enough, will converge to optimal state!

Is this an interesting guarantee?

Sounds like magic, but reality is reality:
- The more downhill steps you need to escape a local optimum, the less likely you are to ever make them all in a row
Simulated Annealing

Theoretical guarantee:
- Stationary distribution: $p(x) \propto e^{E(x)/kT}$
- If $T$ decreased slowly enough, will converge to optimal state!

Is this an interesting guarantee?

Sounds like magic, but reality is reality:
- The more downhill steps you need to escape a local optimum, the less likely you are to ever make them all in a row
- People think hard about ridge operators which let you jump around the space in better ways
Simulated Annealing

Theoretical guarantee:
- Stationary distribution: \( p(x) \propto e^{E(x)/kT} \)
- If T decreased slowly enough, will converge to optimal state!

Is this an interesting guarantee?

Sounds like magic, but reality is reality:
- The more downhill steps you need to escape a local optimum, the less likely you are to ever make them all in a row
- People think hard about ridge operators which let you jump around the space in better ways
Genetic Algorithms

Genetic algorithms use a natural selection metaphor. Keep best N hypotheses at each step (selection) based on a fitness function. Also have pairwise crossover operators, with optional mutation to give variety. Possibly the most misunderstood, misapplied (and even maligned) technique around.
Genetic algorithms use a natural selection metaphor
- Keep best $N$ hypotheses at each step (selection) based on a fitness function
Genetic algorithms use a natural selection metaphor

- Keep best N hypotheses at each step (selection) based on a fitness function
- Also have pairwise crossover operators, with optional mutation to give variety
Genetic algorithms use a natural selection metaphor
- Keep best N hypotheses at each step (selection) based on a fitness function
- Also have pairwise crossover operators, with optional mutation to give variety
Genetic Algorithms

- Genetic algorithms use a natural selection metaphor
  - Keep best N hypotheses at each step (selection) based on a fitness function
  - Also have pairwise crossover operators, with optional mutation to give variety

Possibly the most misunderstood, misapplied (and even maligned) technique around
Example: N-Queens

Why does crossover make sense here?
When wouldn't it make sense?
What would mutation be?
What would a good fitness function be?
Example: N-Queens

Why does crossover make sense here?
Example: N-Queens

Why does crossover make sense here?
When wouldn’t it make sense?
Example: N-Queens

Why does crossover make sense here?
When wouldn’t it make sense?
What would mutation be?
Example: N-Queens

Why does crossover make sense here?
When wouldn’t it make sense?
What would mutation be?
What would a good fitness function be?
Example: Fault Diagnosis

Fault networks:

Variables?

Domains?

Constraints?

Various ways to query, given symptoms

Some cause (abduction)

Simplest cause

All possible causes

What test is most useful?

Prediction: cause to effect

We'll see this idea again with Bayes' nets.
Example: Fault Diagnosis

Fault networks:
- Variables?
- Domains?
- Constraints?

Various ways to query, given symptoms:
- Some cause (abduction)
- Simplest cause
- All possible causes

What test is most useful?

Prediction: cause to effect

We'll see this idea again with Bayes' nets.
Example: Fault Diagnosis

Fault networks:
- Variables?

We'll see this idea again with Bayes' nets.
Example: Fault Diagnosis

Fault networks:
- Variables?
- Domains?
Example: Fault Diagnosis

Fault networks:
- Variables?
- Domains?
- Constraints?
Example: Fault Diagnosis

Fault networks:
- Variables?
- Domains?
- Constraints?

Various ways to query, given symptoms
- Some cause (abduction)
- Simplest cause
- All possible causes
- What test is most useful?
- Prediction: cause to effect

We'll see this idea again with Bayes' nets.
Example: Fault Diagnosis

Fault networks:
- Variables?
- Domains?
- Constraints?

Various ways to query, given symptoms
- Some cause (abduction)
Example: Fault Diagnosis

Fault networks:
- Variables?
- Domains?
- Constraints?

Various ways to query, given symptoms
- Some cause (abduction)
- Simplest cause
Example: Fault Diagnosis

Fault networks:
- Variables?
- Domains?
- Constraints?

Various ways to query, given symptoms
- Some cause (abduction)
- Simplest cause
- All possible causes
Example: Fault Diagnosis

Fault networks:
- Variables?
- Domains?
- Constraints?

Various ways to query, given symptoms
- Some cause (abduction)
- Simplest cause
- All possible causes
- What test is most useful?
Example: Fault Diagnosis

Fault networks:
- Variables?
- Domains?
- Constraints?

Various ways to query, given symptoms
- Some cause (abduction)
- Simplest cause
- All possible causes
- What test is most useful?
- Prediction: cause to effect

We'll see this idea again with Bayes' nets.
Example: Fault Diagnosis

Fault networks:
- Variables?
- Domains?
- Constraints?

Various ways to query, given symptoms
- Some cause (abduction)
- Simplest cause
- All possible causes
- What test is most useful?
- Prediction: cause to effect

We’ll see this idea again with Bayes’ nets.
Beam Search

Like greedy hillclimbing search, but keep K states at all times:

Variables: beam size, encourage diversity?

The best choice in MANY practical settings

Complete? Optimal?

Why do we still need optimal methods?

Greedy Search

Beam Search
Beam Search

Like greedy hillclimbing search, but keep K states at all times:
Beam Search

Like greedy hillclimbing search, but keep K states at all times:
Beam Search

Like greedy hillclimbing search, but keep K states at all times:

Variables: beam size, encourage diversity?
Beam Search

Like greedy hillclimbing search, but keep K states at all times:

Variables: beam size, encourage diversity?

The best choice in MANY practical settings
Beam Search

Like greedy hillclimbing search, but keep K states at all times:

Variables: beam size, encourage diversity?

The best choice in MANY practical settings

Complete? Optimal?
Beam Search

Like greedy hillclimbing search, but keep K states at all times:

Variables: beam size, encourage diversity?
The best choice in MANY practical settings
Complete? Optimal?
Why do we still need optimal methods?
Beam Search

Like greedy hillclimbing search, but keep K states at all times:

Variables: beam size, encourage diversity?
The best choice in MANY practical settings
Complete? Optimal?
Why do we still need optimal methods?
Greedy Search
Beam Search

Like greedy hillclimbing search, but keep K states at all times:

Variables: beam size, encourage diversity?
The best choice in MANY practical settings
Complete? Optimal?
Why do we still need optimal methods?
Greedy Search
Beam Search
CSP Formulation: Fault Diagnosis
CSP Formulation: Fault Diagnosis

Fault networks:
- Variables
- Domains
- Constraints

Various ways to query, given symptoms:
- Some cause (abduction)
- Simplest cause
- All possible causes

What test is most useful?

Prediction: cause to effect

We'll see this idea again with Bayes' nets.
CSP Formulation: Fault Diagnosis

Fault networks:

- Variables?
CSP Formulation: Fault Diagnosis

Fault networks:
- Variables?
- Domains?
CSP Formulation: Fault Diagnosis

Fault networks:
- Variables?
- Domains?
- Constraints?

We'll see this idea again with Bayes' nets.
CSP Formulation: Fault Diagnosis

Fault networks:
- Variables?
- Domains?
- Constraints?

Various ways to query, given symptoms:
- Simplest cause
- All possible causes
- What test is most useful?

Prediction: cause to effect

We'll see this idea again with Bayes' nets.
CSP Formulation: Fault Diagnosis

Fault networks:
- Variables?
- Domains?
- Constraints?

Various ways to query, given symptoms
- Some cause (abduction)

Fault networks with examples:
- SMTP down → Can't email
- DNS down → Can't IM
- Firewall blocking → Can't IM
- Printer jam

We'll see this idea again with Bayes' nets.
CSP Formulation: Fault Diagnosis

Fault networks:
- Variables?
- Domains?
- Constraints?

Various ways to query, given symptoms
- Some cause (abduction)
- Simplest cause
CSP Formulation: Fault Diagnosis

Fault networks:
- Variables?
- Domains?
- Constraints?

Various ways to query, given symptoms
- Some cause (abduction)
- Simplest cause
- All possible causes
CSP Formulation: Fault Diagnosis

Fault networks:
- Variables?
- Domains?
- Constraints?

Various ways to query, given symptoms
- Some cause (abduction)
- Simplest cause
- All possible causes
- What test is most useful?

![Fault network diagram](image)
CSP Formulation: Fault Diagnosis

Fault networks:
- Variables?
- Domains?
- Constraints?

Various ways to query, given symptoms
- Some cause (abduction)
- Simplest cause
- All possible causes
- What test is most useful?
- Prediction: cause to effect
CSP Formulation: Fault Diagnosis

Fault networks:
- Variables?
- Domains?
- Constraints?

Various ways to query, given symptoms
- Some cause (abduction)
- Simplest cause
- All possible causes
- What test is most useful?
- Prediction: cause to effect

We’ll see this idea again with Bayes’ nets.
Next Time: Adversarial Search!

Best strategy against opponent.